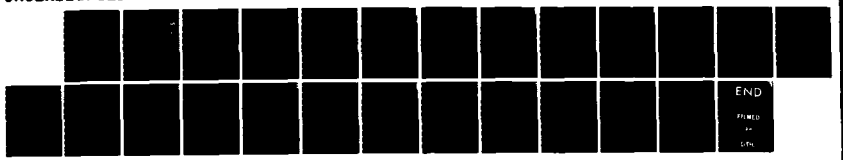
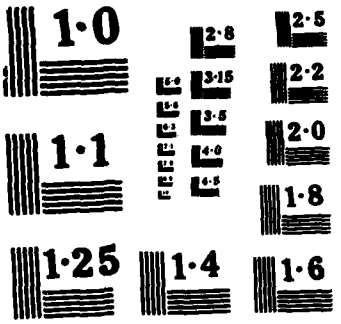
AD-A163 436 COMPACTION WITH AUTOMATIC JOG INTRODUCTION(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE DEPT OF ELECTRICAL
ENGINEERING AND COMPUTER SCIENCE F M MALEY OCT 85
UNCLASSIFIED N00014-80-C-0622 F/G 9/5 NL





DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CAMBRIDGE, MASSACHUSETTS 02139

VLSI Memo No. 85-267 October 1985

Compaction with Automatic Jog Introduction*

F. Miller Maley**

ABSTRACT

This paper presents a novel polynomial-time algorithm for compacting a VLSI layout. Compared to previous algorithms, the algorithm promises to produce higher quality output while reducing the need for designer intervention. The performance gain is realized by converting wires into constraints on the positions of the active devices. These constraints can be solved by graph-theoretic techniques to yield optimal positions for chip components. A single-layer router is then used to restore the wires to the layout, using as many jogs as necessary.

*Published in Proceeding, 1985 Chapel Hill Conference on VLSI, University of North Carolina at Chapel Hill, May 15-17, 1985. This research was supported in part by a Graduate Fellowship from the Office of Naval Research, and by the Defense Advanced Research Projects Agency under contract number N00014-80-C-0622.

**Laboratory for Computer Science and Department of Electrical Engineering and Computer Science, MIT, Room NE43-334, Cambridge, MA 02139, (617) 253-5866.

Compaction with Automatic Jog Introduction

F. Miller Maley
Laboratory for Computer Science
Massachusetts Institute of Technology, Cambridge

Abstract—This paper presents a novel polynomial-time algorithm for compacting a VLSI layout. Compared to previous algorithms, the algorithm promises to produce higher quality output while reducing the need for designer intervention. The performance gain is realized by converting wires into constraints on the positions of the active devices. These constraints can be solved by graph-theoretic techniques to yield optimal positions for chip components. A single-layer router is then used to restore the wires to the layout, using as many jogs as necessary.

1. Introduction

An automated compaction procedure is an effective tool for cutting production costs of a VLSI circuit at low cost to the designer, because the yield of fabricated chips is strongly dependent on the total circuit area. An effective compaction system also reduces design time by freeing the designer from continual concern over design rules. If excess layout space can be removed automatically, the designer can sketch a layout without making constant efforts to conserve area. For these reasons, compaction algorithms have gained widespread attention in the VLSI literature [4, 5, 9, 11], and have been incorporated into many recent computer-aided circuit design systems [2, 4, 10, 17].

Most compaction algorithms, including the one described here, compress a layout in one dimension only. To reduce both dimensions, the layout

This research was supported in part by a Graduate Fellowship from the Office of Naval Research, and in part by the Defense Advanced Research Projects Agency under Contract N00014-80-C-062.

can be alternately compacted in x and y until the algorithm can find no further improvement. Compaction in two dimensions simultaneously is theoretically difficult, although it seems to work well in practice [5]. The remainder of this paper assumes that the direction of compaction is horizontal.

1.1. *Constraint-based compaction*

The more recent systems [4,10] use a *constraint-based* technique to provide one-dimensional compaction. These procedures begin by assigning to each layout component i a variable x_i, which represents the x-coordinate of that component's leftmost point. The *design rules* of the fabrication process are then used to derive constraints on the positions of the components. For example, if device i lies to the left of device j, and such devices must remain at least 2 units apart in order to function reliably, the compactor generates a constraint $x_j - x_i \geq 2 + w_i$, where w_i is the width of component i. (We make the usual assumption that components are not allowed to jump over one another.)

The design rules lead naturally to a set of constraints with very nice properties. First of all, the constraints are not especially difficult to compute [9]. Second, they are sufficient to guarantee that the compacted layout is legal. Third, all constraints are *simple linear inequalities*: they have the form

$$x_j - x_i \geq a_{ij} ,$$

where x_i and x_j are two of the variables assigned to layout components, and a_{ij} is a constant.

Because of the simple form of the inequalities, they can be solved efficiently by graph-theoretic techniques. One constructs an edge-weighted graph whose vertices represent the variables x_i, and in which an edge of weight a_{ij} from node x_i to node x_j represents the constraint $x_j - x_i \geq a_{ij}$. An assignment to the variables x_i that satisfies all the constraints is then determined by a longest-path computation on the graph. The resulting values specify the optimal positions of the components in the compacted layout. A good introduction to constraint-based compaction may be found in [5]; common algorithms for computing longest paths are discussed in [8].

1.2. *Automatic jog introduction*

In order to perform any sort of compaction, the components of the layout must be differentiated into *modules*, which are fixed in size and shape, and *wires*, which are flexible. Common procedures for generating design rule constraints [4,5,9] assume that wires are simply rectangular regions of variable length, and otherwise identical to modules. A vertical wire, for example, would be assigned an x-coordinate during horizontal compaction,

and could only be moved rigidly from side to side. But one would often like a previously straight wire to bend around an obstacle during compaction, if the area of the circuit could thereby be reduced.

This problem is not easily overcome. Many systems [4, 17] attempt to solve it by allowing the designer to specify *jog* points at which wires may bend. Compaction then becomes an interactive procedure in which the designer repeatedly examines the compacted layout, adds more potential jog points, and retries the compaction operation. Other systems [4] attempt to insert jogs automatically, using *ad hoc* techniques which are not guaranteed to be effective. One technique that will work is to insert all possible jog points into every wire. The resulting algorithm has the theoretical drawback that it requires exponential time on some inputs; even in practice, it is likely to consume large amounts of time and memory.

The polynomial-time algorithm presented in this paper has the capability to introduce jog points automatically in an optimal fashion. It can thus be expected to produce high quality output with little designer intervention. Automatic jog introduction is achieved by treating wires not as solid objects, but only as indicators of the topology of the layout. Constraints between modules no longer express design rules directly; instead, they will ensure that there exist paths for the wires, having the given topology, that satisfy the design rules. The new constraints, called *routability* conditions, can be formulated as simple linear inequalities, and solved as usual. When the optimal module placements have been established, the new wire paths are determined by a single-layer router presented in [6]. This router generates no "unnecessary U's," and therefore minimises wire lengths, given that the layout topology is fixed.

This approach to compaction depends on the ability to generate complete routability conditions for a layout. Until recently, such conditions were known only for certain channel routing problems [7,16]. The present work is made possible by the theory of planar routability developed in [1] and [14].

1.3. *Organisation of the paper*

Sect 1 is an introduction.

The remainder of this paper is organised as follows. Section 2 states the definitions and theoretical results that underlie the new compaction method. ~~In this extended abstract, all proofs are omitted.~~ Section 3 shows how the circuit layout is converted to a data structure appropriate for compaction, and Section 4 details the body of the compaction algorithm. ~~I mention in~~ Section 5 several improvements to the algorithm that should make it run considerably faster. Section 6 comments ~~briefly~~ on the algorithm's correctness proof. ~~Finally, I conclude in~~ Section 7 with some extensions of my results, and a discussion of the practical value of the compaction algorithm.

2. Sketches and planar routability

The principal data structure used by the compaction algorithm is called a *sketch*. A sketch represents one layer of a VLSI circuit, including both fixed objects and the topology of the interconnecting wires. The algorithms in this paper process only one sketch at a time, without loss of generality. Compaction of a circuit with multiple layers can be accomplished by computing the constraints for each layer independently, and merging the constraint systems. This section defines precisely what I mean by a sketch, and states the theorem from [6] that determines routability conditions for a sketch. For simplicity, the model we will use is more restricted than necessary. Section 7 notes that some of the restrictions can be relaxed.

2.1. *Definition of a sketch*

A sketch is an ordered pair (F,W) consisting of a finite set F of *features*, which are points and straight line segments, and a finite set W of *wires*, which are simple paths in the plane. Figure 1 shows an example of a sketch. Modules are represented as collections of features, because for technical reasons, terminals must be separated from other features. The features and wires of a sketch must satisfy the following restrictions:

(1) Distinct *components* of the sketch (features and wires) may intersect only at their endpoints.

(2) Distinct wires may not intersect, and no wire may cross itself.

(3) Each wire touches exactly two features, which are single points lying at the endpoints of the wire. They are called the *terminals* of the wire. (Multiterminal wires can also be handled; see Section 7.)

(4) Four of the features of the sketch form a *bounding box* around the other components.

When referring to "points in the sketch," we will mean points lying on features in the sketch. The connected groups of features are called the *obstacles* of the sketch. The definitions imply that each terminal is its own obstacle. Features represent the fixed parts of the layout, the modules; wires represent the flexible interconnections. Clearly, a sketch whose wires are well-behaved (e.g., consist of line segments) can easily be encoded in a data structure.

2.2. *Legality and routability*

I now define what it means to route a sketch, and when a sketch represents a legal layout. A *curve* in a sketch $S = (F,W)$ is a path in the plane that begins and ends on features in F, and intersects no features in between. (For example, the wires in W are curves in S.) Two curves in S are *homotopic* if there is a continuous deformation of the plane that maps one into the other while holding the features F fixed. A *routing* of S is a

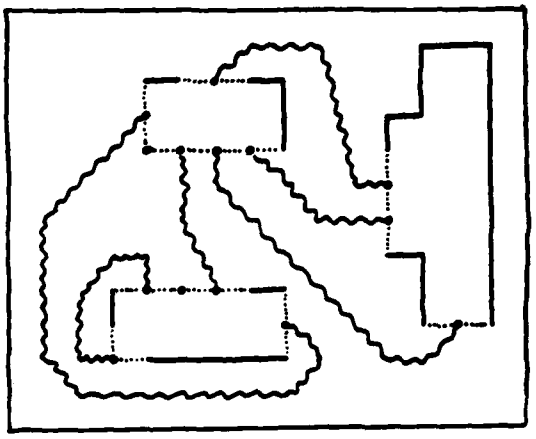


Figure 1. A typical sketch. Solid points and line segments are features, wavy lines are wires, and dotted lines are conceptual module boundaries.

sketch (F, W') whose features are the same, and whose wires can be obtained from the wires W by a continuous deformation of the plane that holds the features F fixed. (Thus the wires in W' are homotopic to the corresponding wires in W.) A sketch is said to be *legal* if it represents a legal VLSI layout, which for our purposes means that the following conditions hold:

(1) All obstacles and wires lie in the rectilinear grid of unit spacing.

(2) The wires form vertex-disjoint paths in the grid.

A sketch is *routable* if it has a legal routing. Using the algorithm in [6], a legal routing of a routable sketch can be found in polynomial time. Figures 1 and 2 illustrate the concepts of legality and routability. The sketch in Figure 1 is illegal, because it contains curved wires. Nevertheless, it is routable, and one of its legal routings is shown in Figure 2.

2.3. *Routability conditions*

The compaction algorithm is based on a theorem from [6] that characterises the routable sketches in terms of the following concepts. If $p = (x_p, y_p)$ and $q = (x_q, y_q)$ are points in the sketch S, then $\overline{pq}$ denotes the open-ended line segment from p to q. Such a segment is called a *cut* if it intersects no features in S. The *capacity* of a cut $\overline{pq}$ is the maximum number of wires that can legally cross $\overline{pq}$; in symbols,

$$cap(\overline{pq}) = \max\{|x_q - x_p|, |y_q - y_p|, 1\} - 1 .$$

The *flow* across $\overline{pq}$, denoted $flow(\overline{pq})$, is the number of crossings of $\overline{pq}$ that are enforced by the topology of the sketch. (See Figure 3.) Crossings of $\overline{pq}$

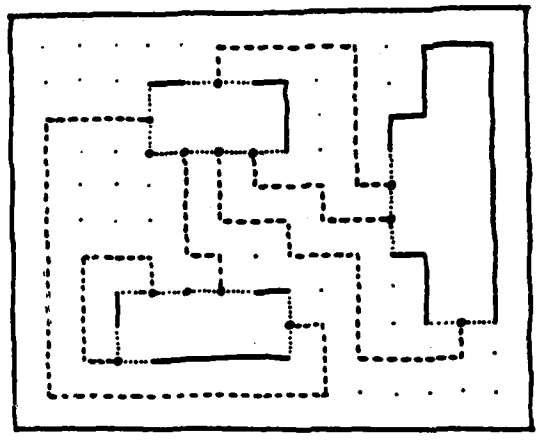


Figure 2. A legal routing of the sketch in Figure 1.

that can be removed by deforming the wires W do not contribute to the flow. More formally, *flow*$(\overline{pq})$ is the minimum, over all routings (F, W') of (F, W), of the number of times $\overline{pq}$ is crossed by wires in W'.

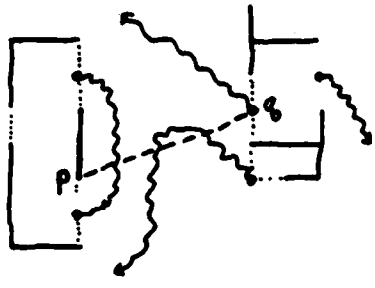


Figure 3. A portion of a sketch with a cut $\overline{pq}$. The flow across $\overline{pq}$ is 1.

The routability of a sketch is completely determined by the flows and capacities of its cuts. Let us say that a cut is *safe* [1] if its flow does not exceed its capacity. Then we have the following result.

Lemma 1. [6] *A sketch that contains an unsafe cut is unroutable.*

More significantly, the converse is true (except when the features of the sketch are illegally placed): a sketch that contains no unsafe cuts is routable.

In fact, this statement may be strengthened. A *critical* cut $\overline{pq}$ is one such that p is the endpoint of a feature, and q is the closest point on its feature to p. The critical cuts are the only important ones.

Theorem 2. [6] The sketch (F,W) is routable if and only if $(F,\emptyset)$ is legal and every critical cut in (F,W) is safe.

The inequalities $flow(\overline{pq}) \leq cap(\overline{pq})$ for the cuts $\overline{pq}$ of a sketch are called the *routability conditions* for the sketch. Constraints of this sort will be used by the compaction algorithm to determine the optimal positions for layout features.

3. Computing flows in the sketch

This section describes a procedure used to facilitate the computation of routability conditions for a sketch. As suggested by Theorem 2, the important attributes of a sketch are the flows and capacities of cuts. Capacities are purely geometric quantities, and can be computed in constant time. In addition, they vary in a regular way with the movement of features during compaction. Flows, on the other hand, are topological quantities, and are relatively difficult to compute. Moreover, they depend in complex ways on the positions of features. Thus to compute flows, we require a data structure that captures the topology of the sketch, and that is invariant under compaction. I begin by presenting such a structure.

3.1. *The adjacency graph*

The data structure we need is called the *adjacency graph* of the sketch. Its construction is straightforward, and is illustrated by Figure 4. From a point on the leftmost edge of each obstacle, except the bounding box, a line is drawn leftward until it hits another obstacle. These line segments and rays will be called *hurdles*. Now each wire is replaced by a homotopic wire that intersects as few hurdles as possible, making sure that no two wires cross. (Wherever a group of wires crosses a hurdle and crosses back, they are moved to the other side of the hurdle.) The resulting set of objects forms a planar graph, whose vertices are obstacles and hurdle/wire crossings, and whose edges are pieces of wires and hurdles. The dual of this graph is the adjacency graph of the sketch. A node of the adjacency graph corresponds to a face of the original graph, and is said to *border* on the points forming the boundary of that face. The adjacency graph does not change during horizontal compaction, because hurdles can only slide back and forth, and we will not allow wires or features to cross over one another.

The purpose of the hurdles is to relate curves in the sketch to the sketch topology. Consider the sequence of hurdles crossed by a curve, in order, together with the directions of crossing. Such a *hurdle sequence* can

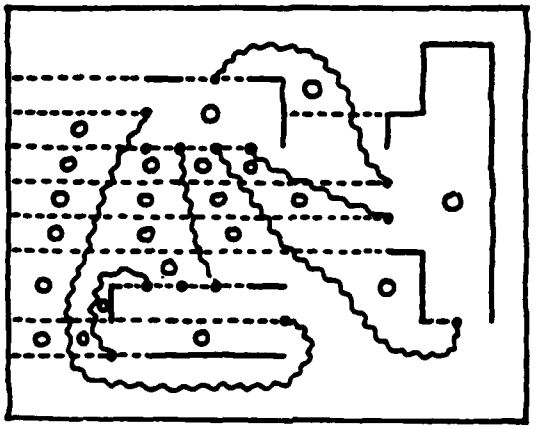


Figure 4. The adjacency graph of the sketch in Figure 1. Dashed lines are hurdles, and circles are nodes of the adjacency graph. Adjacency graph edges are not depicted.

be put into a canonical form by removing all unnecessary crossings, that is, all places where the curve crosses a hurdle and immediately crosses back in the other direction. One can show that two curves with the same endpoints have the same canonical hurdle sequence if and only if they are homotopic.

3.2. *Computation of flows*

An appropriate search through the adjacency graph can compute the flow across a cut. To see how, notice that there are two kinds of edges in the adjacency graph: "wire edges," which represent adjacency across a wire, and "hurdle edges," which represent adjacency across a hurdle. A path through the adjacency graph thus has a hurdle sequence, determined by the hurdle edges it contains. The following lemma demonstrates the correspondence between the two kinds of hurdle sequences. By the *length* of a path in the adjacency graph we mean the number of wire edges the path contains.

> **Lemma 3.** Suppose that a cut $\overline{pq}$ of sketch S has hurdle sequence H. Let $Paths(\overline{pq})$ be the set of paths in the adjacency graph of S, which begin at a node bordering on p, end at a node bordering on q, and have hurdle sequence H. Then $flow(\overline{pq})$ is equal to the length of the shortest path in $Paths(\overline{pq})$.

To make use of this lemma, we must be able to find shortest paths with given hurdle sequences in the adjacency graph. It turns out that the following "greedy" technique suffices. We may assume that each hurdle is to be crossed from bottom to top. Call the adjacency graph G, and define G' to be G with its hurdle edges removed.

Algorithm F. (Computes the flow across a cut.)
Input: a cut $\overline{pq}$ with hurdle sequence $\langle h_1, \ldots, h_n \rangle$, the adjacency graph G.
Local variables: integers i and t, nodes u and v.
Output: the flow f across $\overline{pq}$.

```
1. f ← min{DIST-FROM(w) : w borders on p};
2. function DIST-FROM(w);
3. t ← 0; u ← w;
4. for i ← 1 to n do
     begin
5.     v ← node bordering h_i from below that is closest to u in G';
6.     t ← t + distance from u to v in G';
7.     u ← node adjacent to v across hurdle h_i;
     end;
8. v ← node bordering q from below that is closest to u in G';
9. return t + distance from u to v in G';
```

In other words: for each node bordering on p, find the shortest path to the first hurdle, cross the first hurdle, find the shortest path from there to the second hurdle, and so on. Breadth-first search can be used to implement lines 6–7 and 8–9. This approach may work well in practice, but its worst-case behavior is poor. I show in Section 5 how to implement Algorithm F more efficiently.

4. The compaction algorithm

This section defines mathematically the problem of compaction with automatic jog introduction, and presents a practical algorithm that solves this problem. I assume that each layer of the circuit to be compacted is available in the form of a sketch. In fact, the compaction algorithm will deal with one sketch alone, for the existence of multiple layers adds nothing fundamental to the compaction problem.

4.1. *Configuration space*

Let the input sketch be denoted by S. For the purpose of compaction, the obstacles of S should be grouped into *modules*: collections of features whose relative positions are fixed. The compactor is allowed to choose a horizontal displacement for each module. Such a vector of displacements is called a *configuration* of S. The configuration $\mathbf{d} = \langle d_1, \ldots, d_n \rangle$ corresponds to a sketch in which module i has been shifted right by a distance d_i (or left by a distance $-d_i$). If the sketch has n modules, then the set of all its configurations is the vector space $\mathbf{R}^n$, and the origin $\mathbf{0}$ of this *configuration space* corresponds to the original sketch. The sketch associated with the configuration $\mathbf{d}$ will be denoted $S(\mathbf{d})$, and is uniquely defined (up to

wire homotopy) by requiring that its wires have the same canonical hurdle sequences they had in the original sketch $S(0)$.

Using configurations, we can describe how the sketch changes during compaction. If p is a point in S, its x and y coordinates will be denoted x_p and y_p, respectively. The module in which p lies will be written $\mu(p)$, so the horizontal position of p in the sketch $S(\mathbf{d})$ is $x_p + d_{\mu(p)}$. The notation $p(\mathbf{d})$ stands for p shifted by $\mathbf{d}$, that is, the point $\langle x_p + d_{\mu(p)}, y_p \rangle$. In general, the application of an object to a configuration $\mathbf{d}$ denotes the instantiation of that object in the sketch $S(\mathbf{d})$. Finally, let $\Delta_{pq}(\mathbf{d})$ be difference in x-coordinates between $q(\mathbf{d})$ and $p(\mathbf{d})$; that is,

$$\Delta_{pq}(\mathbf{d}) = (x_q + d_{\mu(q)}) - (x_p + d_{\mu(p)}) .$$

4.2. *Problem statement*

The compaction problem is to find a configuration $\mathbf{d}$ such that $S(\mathbf{d})$ is routable, and can be routed in minimal width. As we have stated it, the compaction problem is generally very difficult; in fact, it is NP-complete [11]. The reason is that the routability conditions may not define a convex region of configuration space, and hence the set of acceptable configurations $\{\mathbf{d} \in \mathbf{R}^n : S(\mathbf{d}) \text{ is routable}\}$ is usually very hard to search. For example, whenever two modules are adjacent, it may be possible to exchange them while maintaining the topology of the sketch, and the resulting sketch may be routable. But intermediate positions, where the modules intersect, are not routable, so the set of acceptable configurations is not convex. Even if we do not expect modules to exchange positions, loops of wire can interfere with one another in the same way, as shown in Figure 5. In most optimization problems, including compaction, one only expects to search a convex subset of the acceptable configurations, in order to achieve a polynomial-time algorithm. The algorithm presented here searches the largest such region that contains the initial configuration, and thus finds the best configuration available to any algorithm of its type.

4.3. *Algorithm overview*

The basic notion underlying the compaction algorithm is that of a *potential cut*. For the purposes of this section, a potential cut is a continuous function that defines for each configuration $\mathbf{d} \in \mathbf{R}^n$ a line segment between two features in $S(\mathbf{d})$. The line segment may or may not be a cut, depending on the positions of the features in $S(\mathbf{d})$. The configuration $\mathbf{c}$ is said to *protect* a potential cut ψ if either $\psi(\mathbf{c})$ is not a cut, or $\psi(\mathbf{c})$ is a safe cut. The significance of these definitions lies in a reformulation of Theorem 2 in terms of potential cuts. That theorem can be read as follows:

> Let $S = (F, W)$ be a sketch, and let $\mathbf{c}$ be a vector in its configuration space. There exist certain *critical* potential cuts, depending

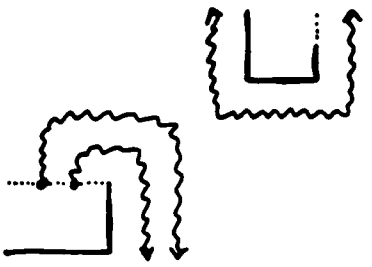


Figure 5. How wires can prevent modules from sliding past one another. If the upper module is allowed to slide past the lower one, the set of acceptable configurations is not convex.

only on the features F, such that the sketch $S(\mathbf{c}) = (F(\mathbf{c}), W(\mathbf{c}))$ is routable if and only if:

(1) the sketch $(F(\mathbf{c}), \emptyset)$ is legal, and
(2) the configuration $\mathbf{c}$ protects every critical potential cut of S.

The compaction algorithm works by finding a subset of configuration space, determined by simple linear inequalities, whose configurations protect every critical potential cut. It thereby ensures that it chooses a configuration that corresponds to a routable sketch. (Condition (1) above can be ignored, because the wireless sketch $(F(\mathbf{c}), \emptyset)$ is legal unless its features fail to lie in the grid. This cannot happen, because the initial sketch (F, W) is assumed to be legal, and the compaction algorithm never considers nonintegral displacements for modules.) The subspace searched is chosen so as to include the initial configuration. An overview of the compaction technique follows.

The central problem is to find a simple linear inequality that ensures that a potential cut, say ψ, is protected. One would like to use the routability condition $cap(\psi(\mathbf{d})) \geq flow(\psi(\mathbf{d}))$ as a constraint on the configuration $\mathbf{d}$, but for most potential cuts ψ, this constraint is not a simple linear inequality. The difficulty lies not with the capacity of $\psi(\mathbf{d})$, which is determined solely by the geometry of $S(\mathbf{d})$, and depends in a simple way on the displacements d_i. Rather, the quantity $flow(\psi(\mathbf{d}))$ is hard to characterize, because it depends on the relation of the line segment $\psi(\mathbf{d})$ to the topology of the sketch $S(\mathbf{d})$.

The solution is to find a specific configuration $\mathbf{c}$ such that whenever the potential cut $\psi(\mathbf{d})$ is unsafe, its flow is equal to $flow(\psi(\mathbf{c}))$. The constraint $cap(\psi(\mathbf{d})) \geq flow(\psi(\mathbf{c}))$ is then sufficient to protect ψ. Moreover, when

this constraint is written in terms of the variables d_i, it becomes a simple linear inequality, because the right hand side is constant. To find $\mathbf{e}$, the algorithm looks for a configuration that minimizes the capacity of ψ, subject to the condition that all critical cuts of smaller vertical span are safe. These shorter cuts force the other features to the side of ψ on which they must lie, if ψ is ever to become unsafe. If, in this way, the algorithm finds a configuration $\mathbf{e}$ that does not protect ψ, the routability condition for $\psi(\mathbf{e})$ is remembered. Otherwise, the potential cut ψ may be ignored.

4.4. *Description of the compaction algorithm*

Since critical cuts move in nontrivial ways during compaction, it turns out to be more convenient to consider the following types of potential cuts:

- Horizontal cuts from feature endpoints.
- Cuts between pairs of feature endpoints.

The constraints generated from these potential cuts turn out to be sufficient to protect all the critical potential cuts.

The horizontal potential cuts are particularly simple, because their flows are independent of the configuration. These potential cuts are treated first in order to generate the constraints that prevent features from crossing over one another. A horizontal potential cut is a function ϕ_{pq} of the form $\phi_{pq}(\mathbf{d}) = \overline{p(\mathbf{d})q(\mathbf{d})}$, where p and q are points in the original sketch. Assuming without loss of generality that $x_q > x_p$, Theorem 2 gives the routability condition

$$\Delta_{pq}(\mathbf{d}) \geq \mathit{flow}(\phi_{pq}(\mathbf{d})) + 1 .$$

Since $\mathit{flow}(\phi_{pq}(\mathbf{d})) = \mathit{flow}(\overline{pq})$ for $\overline{pq}$ horizontal, the constraint is a simple linear inequality

$$d_{\mu(q)} - d_{\mu(p)} \geq (\mathit{flow}(\overline{pq}) + 1) + (x_p - x_q)$$

on the displacements of p and q. The flow across $\overline{pq}$ is easily computed by Algorithm F of the previous section. The initial set of constraints is computed from the horizontal cuts that are incident on feature endpoints. Of course, the constraints are maintained as a graph over the variables d_i. The constraint graph will be called I.

The second stage of the algorithm concerns the cuts that are not horizontal. Let Φ be the set of potential cuts ϕ_{pq} where p and q are feature endpoints with $y_p \neq y_q$. The *height* of an element ϕ_{pq} of Φ is the quantity $|y_p - y_q|$. This quantity is independent of configuration. Sort Φ in increasing order of height, to form a sequence in which flatter potential cuts precede taller ones. After $\phi_{pq} \in \Phi$ has been processed, the algorithm can ensure that the output configuration protects ϕ_{pq}.

The algorithm examines the elements of Φ in sorted order, and for each one that proves important, it adds an appropriate constraint to the graph I. The constraint for a potential cut $\phi_{pq} \in \Phi$, with $x_q \geq x_p$, is computed as follows. First, the algorithm solves the current constraint system I together with the temporary constraint $\Delta_{pq}(\mathbf{d}) \geq 0$, fixing $d_{\mu(p)}$, and minimizing $d_{\mu(q)}$. Call the resulting configuration c. If c protects ϕ_{pq}, then the constraint set is unchanged; otherwise, the constraint

$$d_{\mu(q)} - d_{\mu(p)} \geq (x_p - x_q) + \mathit{flow}(\phi_{pq}(\mathbf{c})) + 1$$

is added to I. The new constraint is a simple linear inequality derived from the routability condition $\mathit{cap}(\phi_{pq}(\mathbf{c})) \geq \mathit{flow}(\phi_{pq}(\mathbf{c})) + 1$.

After all the potential cuts in ψ have been processed, the constraint system I is complete, and the algorithm solves it using a longest-path algorithm. The resulting configuration is used to build an output sketch, which is then routed using a single-layer router such as Algorithm R in [6]. That particular router has the advantage of being able to minimize the lengths of the wires in the routing.

The compaction algorithm is summarized below. We assume that the left and right edges of the sketch's bounding box compose modules 1 and n, respectively, and that the top and bottom edges of the box are ignored.

Algorithm C. (Compacts a sketch in the x-direction.)
Input: a sketch $S = (F, W)$.
Local variables: the points p and q, a configuration c, and the constraint graph I over variables d_i $(1 \leq i \leq n)$.
Output: the compacted sketch.
Subroutines: Algorithm F is used to compute flows in lines 2 and 5; Dijkstra's algorithm is used in lines 4 and 7; Algorithm R (a single-layer router from [6]) is used in line 8.

1. Preprocess S as described in Section 3;
2. Let I be the set of constraints $\{\Delta_{pq}(\mathbf{d}) \geq \mathit{flow}(\overline{pq}) + 1\}$ where $\overline{pq}$ is a horizontal cut with $x_p < x_q$ and either p or q is an endpoint of a feature in F.
3. foreach pair of feature endpoints $\{p, q\}$ with $x_p \leq x_q$ and $y_p \neq y_q$, in order of increasing height do
   begin
4. Find a configuration c that minimizes $c_{\mu(q)} - c_{\mu(p)}$ while obeying the constraints $I \cup \{\Delta_{pq}(\mathbf{d}) \geq 0\}$;
5. if $\overline{p(\mathbf{c})q(\mathbf{c})}$ is a cut in $S(\mathbf{c})$ then if $\mathit{flow}(\phi_{pq}(\mathbf{c})) > \mathit{cap}(\phi_{pq}(\mathbf{c}))$
6. then add the constraint $\Delta_{pq}(\mathbf{d}) \geq \mathit{flow}(\phi_{pq}(\mathbf{c})) + 1$ to I
   end;
7. Find a configuration c satisfying I that minimizes $c_n - c_1$;

8. Route the sketch $S(\mathbf{c})$ and output the result.

4.5. *Details of the implementation*

• The computation in line 2 is easily performed using Algorithm F of the previous section. The cuts themselves may be found by any straightforward method, as the algorithm's run time will be dominated by other factors.

• Line 3 requires that pairs of feature endpoints be enumerated in order of vertical separation. Writing down the pairs and sorting them would waste large amounts of space; the following approach is better. First sort the feature endpoints by y-coordinate, and associate with each endpoint the next higher endpoint. Place these pairs in a priority queue, and keep the queue sorted by difference in y-coordinates. At each iteration of the loop (lines 3–6), withdraw the best element $\{p, q\}$ from the priority queue, and process the potential cut ϕ_{pq}. Then find the next endpoint q' above q in y-coordinate, if one exists, and insert the pair $\{p, q'\}$ into the priority queue. This method uses linear space, and no more time than other parts of Algorithm C.

• To solve the constraint system in line 4, it suffices to compute longest paths from the vertex $\mu(p)$. Dijkstra's algorithm can be used for the purpose, because every edge in the graph has weight zero or less. (Normally, Dijkstra's algorithm is used to find shortest paths, and then the edge weights must be nonnegative.) To see why edge weights are nonpositive, consider the case when all the displacements d_i are zero. Using the assumption that the initial configuration is legal, one can prove that it obeys all constraints. Hence if $d_j - d_i \geq a_{ij}$ is a constraint in I, then it holds under the assigment $\mathbf{d} = \mathbf{0}$. The result is that $0 - 0 \geq a_{ij}$, that is, a_{ij} is nonpositive.[1]

• Once the algorithm finds the key configuration $\mathbf{c}$ in line 4, line 5 must determine whether $\phi_{pq}(\mathbf{c})$ is a cut, and if so, what value $\mathit{flow}(\phi_{pq}(\mathbf{c}))$ takes on. The former possibility may be checked by looking for a feature that intersects $\phi_{pq}(\mathbf{c})$. To compute $\mathit{flow}(\phi_{pq}(\mathbf{c}))$, Algorithm F is invoked. It requires as input the hurdle sequence of $\phi_{pq}(\mathbf{c})$, which can be found by checking every hurdle that lies between y_p and y_q in altitude. Include only

---

[1] This simple observation has important implications for the problem of compaction, with or without jog introduction. Essentially, it says that whenever one's input layout satisfies all the compaction constraints, one can ensure that all the edge weights in the constraint graph have the same sign, just by choosing the proper coordinate system. The compaction variables must represent *offsets* from the initial module positions, or equivalently, the reference point of each module must be placed on the y-axis. Then Dijkstra's algorithm may be used to solve the constraint system, giving asymptotic performance on a graph $G = (V, E)$ of $O(|E| + |V| \log |V|)$ [3], as opposed to $O(|V||E|)$ for the general Bellman-Ford algorithm. See [13] for a discussion of this idea.

those hurdles that cross $\overline{p(c)q(c)}$ in $S(c)$. (A hurdle $\overline{rs}$ transforms to $\overline{r(c)s(c)}$ in $S(c)$, just like any other cut.) The hurdle sequence should, of course, be sorted by y-coordinate, and all crossings must be from bottom to top. Presorting all the hurdles by y-coordinate eliminates the need to sort each individual hurdle sequence.

• In line 7, Dijkstra's algorithm should be used once again, this time computing longest paths in I from module 1, which is the left edge of the bounding box of the sketch. If desired, the designer or design system may add other simple linear inequalities to I, provided that they are all satisfied by the initial layout $S(0)$.

• The configuration c found in line 7 specifies the optimal compacted sketch, but that sketch must still be constructed at line 8. For the purpose of applying Algorithm R, the single-layer wire-router of [6], it is not necessary to construct a complete sketch $S(c)$, but only to produce something called the *rubber-band equivalent* (R.B.E.) of $S(c)$. The features of the R.B.E. are the same as those of $S(c)$, and can be located easily. The wires of the R.B.E. can be found as follows. The set of points not lying on features or hurdles of $S(c)$ is a simply connected region, and its boundary is polygonal (if we allow vertices at infinity). Hence it can be triangulated quickly, and the resulting set of triangles forms a tree under the obvious adjacency relation. We can therefore find for each wire w in $S(c)$ the shortest sequence of triangles that a routing of w could pass through, and apply Algorithm W from [6] to find the wire in the R.B.E. corresponding to w.

4.6. *Complexity analysis*

We evaluate the time performance of Algorithm C in three parts. The notation $|D|$ is used to mean the size of the data structure D. In the worst case, the most time-consuming portions of the algorithm are the $\Theta(|F|^3)$ calls to Dijkstra's algorithm in line 4. Each call takes time $O(|E|+|V|\log|V|)$ on a graph (V, E) [3]. Since $|E|$ is $O(|I|)$, and $|V|$ is n, the number of modules, this leads to an estimate of $\Theta(|F|^3(|I|+n\log n))$ time. Another contributor to the running time is the call to Algorithm F in line 5. If Algorithm F is implemented as suggested in the following section, then each of its $\Theta(|F|^3)$ invocations requires $O(|F|\log|F|\log|G|)$ time, where G is the adjacency graph of the sketch. So this portion of the algorithm takes $O(|F|^3\log|F|\log|G|)$ time. Finally, careful analysis shows that the construction and routing of the output sketch requires only $O(|F||G|\log|G|)$ time [6]. Other operations, including preprocessing the sketch, require considerably less time under the reasonable assumption that $|W|$, the number of wire segments in the input, is $O(|F|^2)$.

Which of the three major parts of Algorithm C will dominate in practice is not clear. In the worst case, $|G|$ can be as high as $\Omega(|F||W|)$, if each of

$\Omega(|W|)$ wire segments intersects a large fraction of the hurdles, and the crossings are not redundant. In most situations, however, $|G|$ should be closer to $|F|$. Making reasonable estimates about the average run time of Algorithm F and the density of the constraint graph I, one can predict that actual performance for the entire operation will probably approach $\Theta(|F|^{3+\epsilon})$ for some small value of ϵ.

Space usage is easier to evaluate: the main contributors are the graphs G and I, along with Algorithm R, which may use $O(|F||G|)$ space in the worst case. Thus the worst case bound is $O(|F|^2|W|)$, but none of the data structures of Algorithm C or Algorithm R is likely to approach its maximum size. The actual figure will depend on the number of crossings between wires and certain cuts in the sketch (*e.g.*, hurdles), and will probably be $\Theta(|F|^{1+\alpha})$ for some constant $\alpha \in (0,1)$.

5. Improvements and optimisations

The compaction algorithm has been stated in its simplest form, and there is considerable room to improve the performance of many of its steps. This section collects several suggestions for speeding up the algorithm. The most important of these ideas is to add a preprocessing phase to Algorithm C that allows flows across cuts to be computed quickly.

5.1. *Faster computation of flows*

Recall that the compaction algorithm computes flows using Algorithm F from Section 3. The most time-consuming steps of that algorithm involve searching through a subgraph G' of the adjacency graph of the input sketch. Algorithm F can be implemented efficiently by taking advantage of the fact that the graph G' is a tree. The first task is to preprocess G' so that one can quickly determine the distance between any pair of its nodes, and hence speed up lines 7 and 9 in Algorithm F. The second task is to preprocess G' so that one can compute efficiently the closest node in a connected subset of G' to a given node. This ability is sufficient to implement lines 6 and 8 of Algorithm F, because the set of nodes bordering a hurdle or feature from below is connected in G'. We wish to minimise the amount of space used by our data structures; in particular, we cannot store explicitly the distance between every pair of nodes.

The solution is to decompose recursively the graph G', and store with a node only the distance to each of its ancestors in the decomposition. Figure 6 shows the construction. Let n be the number of nodes in the graph G'. The separator theorem for trees [12] implies that G' contains a vertex r whose removal disconnects G' into subtrees containing at most $\frac{2}{3}n$ nodes each. Moreover, the vertex r can be found in linear time using depth-first search

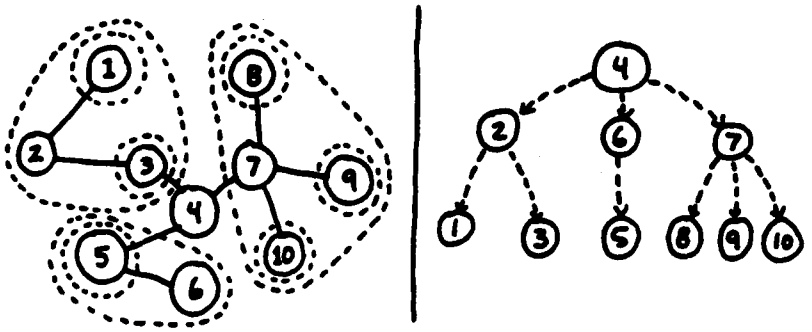


Figure 6. A tree, drawn with solid lines, and its decomposition tree.

to compute the sizes of subtrees. If we store with every node its distance from r, then the distance between two nodes in different subtrees of r can be computed by summing their distances from r. Now we decompose the subtrees of r recursively, forming a *decomposition tree* D on the nodes of G'. At each stage of the decomposition, the sizes of subtrees are reduced by a constant factor, so D has logarithmic height. To find the distance between two vertices in G', one first finds their lowest common ancestor (LCA) in D, and then sums their distances from that ancestor. This procedure clearly takes at most $O(\log |G|)$ time.

Some extra preprocessing is needed before we can compute closest members of connected sets of G. Let D be the same decomposition tree as above. The LCA in D of a connected set $C \subseteq G'$ is a member of C, and we can compute in advance the LCA's of the connected sets that we care about. We also store, for each node y and for each of its ancestors z, the highest vertex in D that is interior to the to the path between z and y in G'. In case z and y are adjacent in G', we store *nil* instead.

I now describe how to compute the closest vertex in a connected set $C \subseteq G'$ to a node u. If u lies in C, then the answer is u. Otherwise, set v to be the LCA of C. If v is not an ancestor of u, then replace u by the LCA of u and v. Now perform a "binary search" on the path between u and v as follows. Let z be the highest vertex in D between u and v. If z is *nil* then the answer is v; otherwise set $v \leftarrow z$ or $u \leftarrow z$ according to whether z does or does not lie in C, and repeat the process. After at most $O(\log |G|)$ iterations, the search will terminate, and each iteration can be made to run in $O(\log |C|)$ time.

5.2. *Compressing the adjacency graph*

Both the time and space performance of Algorithm C can be improved

by reducing the size of the adjacency graph. The trick is to choose hurdles in such a way as to minimize the number of crossings between wires and hurdles. Although we defined hurdles so that every obstacle has only one hurdle incident on its left, this property is unimportant. The hurdles can be chosen to be any set of horizontal cuts such that the set of points inside the bounding box, but not lying on a hurdle or a feature, is simply connected. Equivalently, if obstacles and hurdles are considered as the nodes and edges, respectively, of a graph, then this graph must be a tree.

We can use a minimum-cost spanning tree algorithm to find a set of hurdles that cross as few wires as possible. Every horizontal cut between different obstacles is a potential hurdle, but we may restrict our attention to horizontal cuts that are incident on feature endpoints. There are at most $O(|F|)$ such cuts, and they can be thought of as the edges of a graph H over the obstacles. The cost of an edge will be the number of crossings of the cut by wires in the original sketch; costs can be computed efficiently using a scanning algorithm as in [6]. The hurdles are chosen to be the edges in a minimum-cost spanning tree of the graph H.

5.3. *Ignoring unimportant potential cuts*

A few simple checks will probably speed up Algorithm C by a factor of 4 or more. First of all, a corollary of the correctness proof is that a potential cut ϕ_{pq} whose capacity is minimal in the initial configuration cannot generate a constraint. Therefore, the algorithm need only check potential cuts ϕ_{pq} for which $|x_q - x_p| > |y_q - y_p|$. Second, the lower endpoint of a feature need not be considered in conjunction with feature endpoints above it, and symmetrically for the upper endpoint. Similarly, a potential cut which in the initial configuration travels rightward from the left endpoint of a horizontal feature need not be considered, and symmetrically for right endpoints. Finally, a potential cut ϕ_{pq} with $x_q > x_p$ need not be checked if in all configurations d such that $\Delta_{pq}(d) > 0$, the line segment $\phi_{pq}(d)$ is not a cut. This is known to be true, for example, if every rightward cut from p crosses a feature in the same module as p.

6. Highlights of the correctness proof

The correctness proof of Algorithm C cannot be included here due to space restrictions, for it is highly involved. It does, however, lead to an understanding of the assumptions that underlie the compaction algorithm. In this section I state a set of conditions under which one can construct a compaction algorithm that performs automatic jog introduction in the manner of Algorithm C. Knowing these conditions, one can determine whether Algorithm C will continue to work if the definition of a legal sketch is modified.

We require a more technical definition of potential cut. Let P and Q be features in the original sketch S, and let ψ be a continuous function that defines, for each configuration $\mathbf{d}$, a line segment between the features $P(\mathbf{d})$ and $Q(\mathbf{d})$ in the sketch $S(\mathbf{d})$. The function ψ is a potential cut if the position of $\psi(\mathbf{d})$ relative to $P(\mathbf{d})$ and $Q(\mathbf{d})$ depends only on the displacement between $P(\mathbf{d})$ and $Q(\mathbf{d})$, namely $\Delta_{PQ}(\mathbf{d}) = d_{\mu(Q)} - d_{\mu(P)}$ (stretching the notation slightly). In other words, ψ must satisfy the following condition.

> If $\mathbf{d}$ and $\mathbf{d}'$ are any two configurations such that $\Delta_{PQ}(\mathbf{d}) = \Delta_{PQ}(\mathbf{d}')$, then $\psi(\mathbf{d}')$ is equal to $\psi(\mathbf{d})$ shifted to the right by $d'_{\mu(P)} - d_{\mu(P)}$ units.

Algorithm C compacts a sketch S by computing a sequence $\Phi(S) = \langle \psi_1, \ldots, \psi_m \rangle$ of potential cuts of S, and using their routability conditions to constrain the positions of modules in S. The proof that this technique generates a sufficient and optimal constraint system depends on several facts. One such fact is Lemma 1, which ensures that an unsafe cut makes a sketch unroutable. Another important fact is that the potential cuts $\Phi(S)$ determine the routability of the modified sketches $S(\mathbf{d})$. Specifically, they have the following property:

> *Routability property.* If $S(\mathbf{0})$ is routable, and for all $\lambda \in [0,1]$ the configuration $\lambda\mathbf{d}$ protects every $\psi \in \Phi(S)$, then $S(\mathbf{d})$ is routable.

The capacities of the potential cuts also have a special property:

> *Bitonic property.* For each $\psi \in \Phi(S)$, and each line L in configuration space, there is a point $\mathbf{c}$ of L at which the capacity $cap(\psi(\mathbf{c}))$ is minimal, and $cap(\psi(\mathbf{d}))$ is nondecreasing as $\mathbf{d}$ moves away from $\mathbf{c}$ along L.

In principle, our compaction method depends on only one further fact.

> *Ordering property.* Suppose that the following statements hold.
> (1) The configuration $\mathbf{d}$ protects ψ_i for all $i < k$.
> (2) The configuration $\mathbf{d}$ lies on the boundary of the set $\{\mathbf{c} \in \mathbf{R}^n : \psi(\mathbf{c}) \text{ is a cut}\}$.
> (3) The cut s is properly contained in the line segment $\psi_k(\mathbf{d})$.
>
> Then s is safe in the sketch $S(\mathbf{d})$.

Algorithm C satisfies this requirement by sorting its potential cuts according to height.

The conditions listed above allow us to determine when Algorithm C can be extended to more general kinds of layouts. Suppose that under new definitions of "sketch" and "legal," Lemma 1 continues to hold, and that given a legal sketch S, one can compute in polynomial time a sequence $\Psi(S)$ of potential cuts with the routability, bitonic, and ordering properties.

Suppose also that one has a single-layer router that can fill in the wires of S after compaction. Then the technique of Algorithm C gives a polynomial-time algorithm to compact the sketch S.

7. Extensions and discussion

This section suggests several ways in which the compaction algorithm can be extended, and concludes with an overview of the main results.

7.1. *Generalizations of Algorithm C*

7.1.1. *Multiple layers.* We have assumed that the layout to be compacted comprises only one layer. To remove this restriction, Algorithm C need only generate constraints for each layer of the design independently, as in lines 1–6. Since most modules have components on more than one layer, the resulting constraint systems must be merged, by choosing the most restrictive constraint between every pair of modules. The merged system is then solved normally.

7.1.2. *Wire widths.* None of my results depend on the assumption that all wires have the same width. If they do not, one need only replace the counting of wires by the summation of their widths, in the definition of flow, in Lemma 3, and in Algorithm F. With suitable modifications, features can have differing thicknesses as well. More details are included in [14].

7.1.3. *Multiterminal nets.* One major deficiency of the sketch as a layout model its limitation on the number of terminals that may contact a wire. The model can be generalized, however, and the compaction algorithm remains essentially unchanged. (One difference is that Algorithm R, the single-layer router, can no longer minimize the wire lengths in the output sketch.) See [6] for a discussion of how to handle multiterminal wires.

7.2. *Summary and conclusion*

The main theoretical contribution of this paper is a polynomial-time algorithm that compacts IC (or PCB) layouts while introducing jogs into wires in an optimal fashion. The power of Algorithm C comes from the elimination of wires as hard objects in the layout, and their replacement by constraints between modules. The use of routability conditions to solve placement problems is not new [7,15,16], but until now, only channel routing problems had been considered. The reason is that the routability of general planar layouts was not adequately understood until very recently [1,6]. To characterize planar routability requires a robust model of a circuit layer, such as the sketch, and a fair amount of theory. In addition, some care is needed to apply routability conditions to the compaction of general sketches; proving the correctness of Algorithm C is nontrivial.

On the practical side, my compaction method can be expected to produce high-quality layouts with little designer intervention, saving both in chip area and design time. Its primary drawback lies in its use of computational resources. Although there are good reasons to believe that its worst-case performance bounds will not be approached in practice, resource limitations may prevent it from being used to compact large layouts all at once. Algorithm C is amenable to use at all levels of the design, however, so that hierarchical compaction can alleviate much of the resource problem. It also may be suited to use in channel routing, where the number of components is not too great. The idea, which was implemented at Bell Labs (see Acknowledgements) is as follows: the channel is artificially inflated, so that an an ordinary channel routing algorithm, which may have difficulty with crowded channels, may succeed; then a compactor like Algorithm C, with the ability to insert arbitrarily complex jogs, is applied in order to compact the channel back to the proper size.

One important question left open by my research is whether the compaction method embodied in Algorithm C is more efficient in practice than the straightforward algorithm, namely, inserting jog points into each wire where it crosses each horizontal gridline, and solving the resulting constraint system normally. This technique is evidently simpler than that of Algorithm C, and may be more efficient in practice. On the other hand, it may be possible to extend Algorithm C to situations where wires and modules may contain diagonal segments, and grid-based algorithms break down.

Acknowledgements

I would like to thank my advisor Charles Leiserson for many helpful discussions, and for corrections and comments on this paper. The starting point of this research was provided by Prof. Leiserson who, along with his former student Ron Pinter, discovered the connection between routability conditions and compaction with automatic jog introduction. Some special case algorithms they developed were implemented by Andrew Hume at AT&T Bell Laboratories, and used for channel routing. Thanks also to Ron Greenberg for comments on a draft of this paper.

References

[1] R. Cole and A. Siegel, "River routing every which way, but loose," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science* (October 1984), pp. 65–73.

[2] A. E. Dunlop, "SLIP: symbolic layout of integrated circuits with compaction," *Computer Aided Design*, Vol. 10, No. 6 (November 1978), pp. 387–391.

[3] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Proceedings of the 25th Annual Symposium on Foundations of Computer Science* (October 1984), pp. 338–346.

[4] M. Y. Hsueh, *Symbolic Layout and Compaction of Integrated Circuits*, Ph.D. thesis, EECS Division, University of California, Berkeley, CA (1979).

[5] G. Kedem and H. Watanabe, "Optimization techniques for IC layout and compaction," Technical Report 117, Computer Science Department, University of Rochester (September 1982).

[6] C. E. Leiserson and F. M. Maley, "Algorithms for routing and testing routability of planar VLSI layouts," to appear in *17th Annual ACM Symposium on Theory of Computing* (May 1985).

[7] C. E. Leiserson and R. Y. Pinter, "Optimal placement for river routing," *SIAM Journal on Computing*, Vol. 12, No. 3 (August 1983), pp. 447–462.

[8] C. E. Leiserson and J. B. Saxe, "A mixed-integer linear programming problem which is efficiently solvable," *Proceedings of the 21st Annual Allerton Conference on Communication, Control, and Computing* (October 1983), pp. 204–213.

[9] T. Lengauer, "Efficient algorithms for the constraint generation for integrated circuit layout compaction," *Proceedings of the 9th Workshop on Graphtheoretic Concepts in Computer Science* (June 1983).

[10] T. Lengauer and K. Melhorn, "The HILL system: a design environment for the hierarchical specification, compaction, and simulation of integrated circuit layouts," *Proceedings, Conference on Advanced Research in VLSI* (January 1984).

[11] T. Lengauer, "On the solution of inequality systems relevant to IC layout," *Proceedings of the 8th Workshop on Graphtheoretic Methods in Computer Science*, Hanser Verlag, München (1982).

[12] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal on Applied Mathematics*, Vol. 36, No. 2 (April 1979), pp. 177–189.

[13] F. M. Maley, "An observation concerning constraint-based compaction," in preparation.

[14] F. M. Maley, *Compaction and Single-Layer Routing*, M.S. Thesis, MIT Department of Electrical Engineering and Computer Science (to be submitted May 1985).

[15] R. Y. Pinter, *The Impact of Layer Assignment Methods on Layout Algorithms for Integrated Circuits*, Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science (August 1982).

[16] A. Siegel and D. Dolev, "The separation for general single-layer wiring barriers," *Proceedings of the Carnegie-Mellon Conference on VLSI Systems and Computations* (October 1981), pp. 143–152.

[17] J. D. Williams, "STICKS – a graphical compiler for high level LSI design," *National Computer Conference* (1978), pp. 289–295.

END

FILMED

2-86

DTIC